PAPER

An Instruction Mapping Scheme for FU Array Accelerator

Kazuhiro YOSHIMURA^{†a)}, *Student Member*, Takuya IWAKAMI[†], Takashi NAKADA[†], *Nonmembers*, Jun YAO[†], Hajime SHIMADA[†], *Members*, and Yasuhiko NAKASHIMA[†], *Nonmember*

SUMMARY Recently, we have proposed using a Linear Array Pipeline Processor (LAPP) to improve energy efficiency for various workloads such as image processing and to maintain programmability by working on VLIW codes. In this paper, we proposed an instruction mapping scheme for LAPP to fully exploit the array execution of functional units (FUs) and bypass networks by a mapper to fit the VLIW codes onto the FUs. The mapping can be finished within multi-cycles during a data prefetch before the array execution of FUs. According to an HDL based implementation, the hardware required for mapping scheme is 84% of the cost introduced by a baseline method. In addition, the proposed mapper can further help to shrink the size of array stage, as our results show that their combination becomes 88% of the baseline model in area.

key words: instruction mapping, FU array, coarse-grained reconfigurable architecture

1. Introduction

Recently, high-performance processors are desired in many areas such as high-performance computing and image processing, which require large amounts of computation. Accordingly, cooling has become a critical requirement for computer systems that contain such processors. A common solution for this problem is to use Application Specific Integrated Circuits (ASICs), which provide high-performance and low-power features targeted for individual programs. However, developing ASICs can not meet product cycle because of their high economical cost and long development period.

Alternatively, many studies have proposed relieving the development cycle problem of ASICs by using techniques such as Many-Core Architectures (MCAs) [1], [2] and Coarse-Grained Reconfigurable Architectures (CGRAs) [3]. MCAs are composed of general purpose processors (GPPs), which are able to execute conventional Instruction Set Architecture (ISA) based machine instructions and thus provide high-programmability with the support of existing compilers. However, in contrast to ease of programmability, the power consumption of MCAs increases in proportion to the number of GPPs contained in a processor. Therefore, the energy efficiency of MCAs is limited by the number of GPPs. Figure 1 shows relative ratings of current implementations in terms of energy/area efficiency based on a research

Fig. 1 The position of our processor and today's implementation from the view of energy and area efficiency on 130 nm CMOS technology [4].

by Tobias, et al. [4]. GPP and GPP-based MCAs have relatively low efficiency as compared to other specifically purposed implementations. Opposite to MCAs, CGRAs obtain better performance per power consumption by reconfiguring the interconnection between many functional units (FUs). However, in contrast to its good energy efficiency, CGRAs usually lack programmability, and thus require a special compiler [5], [6] to generate configuration information. This is similar to logic synthesis tools and a special ISA [7], [8], which explicitly specifies data flows between FUs.

As an alternative to these approaches, we have proposed a programmable and power-efficient processor named the Linear Array Pipeline Processor (LAPP) [9], [10] as shown in Fig. 1. An LAPP has a traditional VLIW pipeline and an FU array pipeline like CGRAs. An LAPP can directly execute existing programs that are compiled for traditional VLIW processors, on the traditional VLIW pipeline. An LAPP can also execute a loop kernel on the FU array pipeline efficiently with array executions. The loop kernel is mapped onto FUs and configures interconnection between the FUs and registers. Specifically, an LAPP works by assigning one VLIW instruction to one stage in the FU array pipeline during the array execution. This execution model can help achieve sufficient acceleration with minimal required resources. The gating of unnecessary instruction cache and FUs can grant an effective power saving in a sufficiently long term mapping.

To achieve high programmability, a mechanism is required to ensure that every FU can read appropriate data every clock cycle. One straightforward way is to clone the register file for every stage. However, placing a cloned register file per stage leads to a huge increase in the circuit

Manuscript received June 17, 2010.

Manuscript revised October 9, 2010.

[†]The authors are with the Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630-0192 Japan.

a) E-mail: kazuhiro-y@is.naist.jp

DOI: 10.1587/transinf.E94.D.286

area and critical path delay. To avoid this overhead, our proposed mapping scheme uses register renaming from architectural register to reduce the input registers of execution units. The proposed mapping scheme can help execute traditional VLIW instructions on FU array processors. The instruction mapper maps the instructions onto FUs and configures their interconnection. The circuit area and delay time of the mapper have been extensively studied to avoid large hardware extension, with the help of a simple register renaming method.

The rest of this paper is organized as follows. Section 2 explains an outline of our baseline processor named LAPP as an implementation of FU array processor. Section 3 details the instruction mapping scheme for such processors. The hardware cost is evaluated in Sect. 4 by studying the circuit area and the delay time of each module in LAPP. Section 5 describes related work. Section 6 summarizes this paper.

2. Outline of LAPP

This section introduces LAPP architecture, which serves as the target platform of this research. As introduced in Sect. 1, LAPP targets at high-performance and high energy efficiency while maintaining good programmability by keeping backward compatibility with traditional processors. The following ideas are introduced in LAPP.

- LAPP is based on a VLIW processor for high parallelism computation such as image processing.
- LAPP has the extended backend pipeline of a traditional VLIW pipeline linearly to maintain compatibility to traditional processors. FU array consists of the backend pipeline. It includes both execution units and memory access units.
- The instruction mapper of LAPP maps all of VLIW instructions in a loop kernel onto FU array.
- Data prefetching by a prefetch instruction guarantees cache hit of load/store (LD/ST) operations during array execution.

On the basis of above ideas, LAPP also has the following constraints.

- The number of stages in FU array is fixed when the processor is designed. It limits the number of VLIW instructions in a loop kernel.
- The size of input data accessed during array execution is smaller than the size of the level 1 data cache. Similarly, the size of input data accessed from an iteration is smaller than the size of local cache memory.
- Each iteration can only store data within 1 word (32-bit).
- No data or memory dependence is allowed across loop iterations as the loop kernel is well pipelined into FU array stages and the executions of iterations are largely overlapped to exploit extreme parallelism.
- A loop kernel includes an unconditional backward

Fig. 2 Modules and connections in an LAPP.

branch instruction to continue the execution and a conditional forward branch instruction to terminate the execution.

LAPP achieves high IPC by mapping VLIW instructions onto the FU array efficiently. LAPP has 3 execution modes: Normal-Execution, Array-Setup and Array-Execution.

During Normal-Execution, the VLIW pipeline only executes VLIW instructions as a traditional VLIW processor and the array pipeline is halted. Meanwhile, in Array-Execution, LAPP additionally uses FU array to exploit parallelism. Array-Setup is invoked when a special prefetch instruction is detected. In Array-Setup, instruction mapper starts mapping VLIW instructions in a loop kernel onto FUs. The instruction mapper also starts configuring the interconnection between FUs by arranging multiplexers so as to forward values of registers from one stage to its succeeding stages. Data prefetching between the level 1 data cache and level 2 unified cache is started at the same time. Using this overlapping, the overhead of mapping and configuring can be hidden by the prefetch delay, as the data prefetching is usually longer than mapping and configuring.

After all of mapping, configuring and prefetching are completed, LAPP invokes Array-Execution. Based on the well mapped VLIW instructions and interconnections, Array-Execution can process these instructions in a highly parallel fashion. Data from the level 1 data cache flows into the level 0 data cache, which is a kind of local memory for each stage, toward the following stages. The interconnection for forwarding the contents of registers and the level 0 cache is maintained in Array-Execution. According to this design, LAPP can produce the result of one loop iteration per clock cycle. The result data are stored in the level 1 cache.

We illustrate the modules and connections in an LAPP in Fig. 2. LAPP is constructed with a traditional VLIW pipeline and FU array pipeline. Figure 2 shows stage 0 as the traditional VLIW pipeline and, stage 1 and stage 2 as the FU array pipeline. The VLIW pipeline contains a level 1 data cache (L1\$), a level 1 instruction cache (I1\$), a

program counter (PC), an instruction fetch unit (IF), an instruction decoder (ID), a register file (RF), several execution units (EXEC) and a memory access unit (MA). The VLIW pipeline also has forwarding data paths (A) and (B) from outputs of the execution units and the memory access unit to the register file. In this VLIW pipeline, the level 0 data cache (L0\$) and instruction mapper (MAP) are used only in Array-Execution.

The FU array pipeline is constructed by extending the backend pipeline of the VLIW pipeline linearly. Each stage is connected to the succeeding stage through pipeline registers. Each stage uses a local level 0 data cache (L0\$) and propagating registers instead of L1\$ and RF in stage 0, respectively. The instruction mapping scheme is performed by a MAP and multiplexers (SEL). The MAP maps VLIW instructions onto the FU array and configures the interconnection between FUs with the SEL. We will describe the details of the instruction mapping scheme in Sect. 3.

Since the register file is only included in the VLIW pipeline which works on traditional VLIW codes, there is no extra hardware extension such as additional read/write ports in it. The number of LAPP array stages depends on the target applications. As one VLIW instruction is mapped onto one stage, the required stages are equal to the number of VLIW instructions in the loop kernel that will be executed in Array-Execution. During the design phase, we can estimate the amount of hardware resources and the effective performance for a target application.

It is possible to design a data path (C) from stage 2 to stage 0 in Fig. 2 to store the updated intermediate values into the register file after array execution. However, in most cases, these intermediate values stored in propagation registers such as loop counters are not needed for program execution beyond the loop kernel. For this reason, data path (C) is currently not appeared in the implementation.

In LAPP, unlike a text book pipeline which usually contains only one memory access stage, the array pipeline to perform all of the instructions in one loop iteration may include several memory access stages. To reduce the port number of the cache memory, LAPP uses a local cache memory for each array stage. During array execution, according to LAPP's design, array stages work on different loop iterations. Therefore, the data in different local cache memories must be propagated in the iteration order along the array stage propagating direction as shown in Fig. 2. As a result, LAPP can simultaneously handle many memory access requests from array stages based on this cache structure.

This structure requires the supports from programmers or compilers to guarantee that there are no register and memory dependencies between the iterations in a loop, when prefetch instructions are inserted to initiate array executions.

As mentioned above, LAPP has machine-instruction level compatibility, programmability based on traditional compiler techniques, and performance improvement with good prospects.

3. Instruction Mapping Scheme

The instruction mapping scheme is one of the most important units for LAPP to efficiently map VLIW instructions onto the FU array and configure interconnections between FUs. In this section, we focus on 3 evaluation models and detail the instruction mapping scheme for FU array processors.

3.1 Execution and Evaluation Model

To execute programs correctly on FU array processors, there are 2 execution models. One is a straightforward model where each stage has a clone register file to forward all values of the register file (RF model). The other is a proposed model where each stage has a limited number of registers to propagate the register values (SEL model). Figure 3 shows three forwarding patterns of RF and SEL models. They are used as evaluation models in Sect. 4.

In the RF model, pipeline registers between adjacent stages can be used to forward a value from one stage to its succeeding stage. However, when forwarding a value from a stage to other non-adjacent stages, routes going through the cloned register files will be used to passing data. This brings two problems. One problem is that forwarding values from stage i (i : stage number) to the cloned register files in stage $(i + X)$ ($X \geq 2$) may lead to a long delay by going through the register files. The other problem is that it requires high-density interconnections between the register files to maintain the logically correct value.

In the SEL model, the instruction mapper renames the architectural register numbers to a small number of propagation registers. Due to the small number, the delay is shortened and the interconnections are reduced. Basically, the

Fig. 3 Execution models and forwarding patterns.

Fig. 4 Evaluation models.

propagation registers can utilize the EXEC input registers that have no mapped instructions. In the worst case, all values of the register file in stage 0 may need to be propagated. However, our previous work [9] has shown that diverting 11 EXEC input registers to propagation register is sufficient to execute useful scientific computation and image processing programs. The configuration of the previous work was as follows. During instruction mapping, if a deficiency of the propagation registers occurs, LAPP discontinues Array-Setup and restarts Normal-Execution.

- Stage 0 includes general and media register files (RFs). The general RF has 11 read ports and 5 write ports, and contains 32 general registers for integer RISC instructions. The media RF has 7 read ports and 4 write ports, and contains 32 media registers for multimedia SIMD instructions.
- Each stage contains 3 arithmetic and logic units (ALUs), an effective address generator (EAG) for load/store (LD/ST), a branch unit (BRC) and 4 media calculation units (MEDIAs) per stage.

In this paper, each stage has the same configuration as the previous work [9]. We evaluate three models focusing on the general RF. The models of RF32 (baseline), RF11 (preliminary) and SEL11 are demonstrated in Fig. 4 (a), (b) and (c) respectively.

RF32 in Fig. 4 (a) has a clone register file with 32 general registers. The data from general registers and EXEC output registers in the previous stage are merged by 32 6-input 1-output multiplexers (6-1 MUX) and assigned to general registers in RF32. The EXEC input registers can get any value from the general registers by a 37-input 1-output multiplexers (37-1 MUX). The contents of the newly updated register file of stage i are cloned to the register file of stage $(i - 1)$ based on a one-to-one relationship.

RF11 in Fig. 4 (b) has a clone register file of a reduced size—11 registers for all general registers. It requires a crossbar switch to clone any content of general registers in the register file. Each EXEC input register can get a value from the general registers by a 16-input 1-output multiplexers (16-1 MUX).

SEL11 in Fig. 4 (c) has 11 registers for register value

propagation. It does not require general registers by diverting 11 EXEC input registers to propagation registers.

According to the number of input and output ports of each module in Fig. 4, we can get the number of interconnections in these three models, as the following calculation. Here, $WIRE_{RF32}$, $WIRE_{RF11}$ and $WIRE_{SEL11}$ are used to denote the number of interconnections.

$$\begin{aligned} WIRE_{RF32} &= 32\text{bit} \times 32(\# \text{ of general register}) \\ &\quad + 32\text{bit} \times 5(\# \text{ of EXEC output}) \\ &\quad + 32\text{bit} \times 5(\# \text{ of EXEC output}) \\ &= 1344\text{wires} \end{aligned}$$

$$\begin{aligned} WIRE_{RF11} &= 32\text{bit} \times 11(\# \text{ of general register}) \\ &\quad + 32\text{bit} \times 5(\# \text{ of EXEC output}) \\ &\quad + 32\text{bit} \times 5(\# \text{ of EXEC output}) \\ &= 672\text{wires} \end{aligned}$$

$$\begin{aligned} WIRE_{SEL11} &= 32\text{bit} \times 11(\# \text{ of EXEC input}) \\ &\quad + 32\text{bit} \times 5(\# \text{ of EXEC output}) \\ &= 512\text{wires} \end{aligned}$$

i.e.,

$$WIRE_{RF32} > WIRE_{RF11} > WIRE_{SEL11}.$$

Similar to the number of wires, $DELAY_{RF32}$, $DELAY_{RF11}$ and $DELAY_{SEL11}$ represent the preceding delay before each ALU. $DELAY_{RF32}$ comes from the 32-1 MUX in Fig. 4 (a). $DELAY_{RF11}$ is the longest one between the crossbar switch and the 16-1 MUX in Fig. 4 (b). $DELAY_{SEL11}$ corresponds to the crossbar switch in Fig. 4 (c). Accordingly, the delay times show the following relationship.

$$DELAY_{RF32} > DELAY_{RF11} = DELAY_{SEL11}$$

Focusing on the same scheme of mapping VLIW instructions onto the FUs and configuring the interconnection between FUs, RF32 does not require the instruction mapper (MAP) due to the full clone of the register file. However, RF11 and SEL11 require MAP to select necessary values from the RF subset. MAP decides necessary values from the

previous stage to the execution unit in each stage and configures interconnection between them. Accordingly, the area of these models, which are defined as $AREA_{RF32}$, $AREA_{RF11}$ and $AREA_{SEL11}$, can be calculated as follows.

$$\begin{aligned} AREA_{RF32} &= 32AREA_{6-1MUX} + 32AREA_{gr} \\ &\quad + 11AREA_{37-1MUX} + 11AREA_s \\ AREA_{RF11} &= 11AREA_{16-1MUX} + 11AREA_{gr} \\ &\quad + 11AREA_{16-1MUX} + 11AREA_s \\ &\quad + AREA_{MAP} \\ AREA_{SEL11} &= 11AREA_{16-1MUX} + 11AREA_s \\ &\quad + AREA_{MAP} \end{aligned}$$

Where $AREA_{6-1MUX}$, $AREA_{16-1MUX}$, $AREA_{37-1MUX}$, $AREA_{gr}$ and $AREA_s$ refer to the circuit area of a 6-1 MUX, a 16-1 MUX, a 37-1 MUX, a general register (gr) and an EXEC input register (s) in Fig.4 respectively. In addition, $AREA_{MAP}$ refers to the circuit area of MAP.

These result the following relationship of the circuit area between RF32 and SEL11.

$$\begin{aligned} AREA_{RF32} - AREA_{SEL11} &= 32AREA_{6-1MUX} + 32AREA_{gr} \\ &\quad + 11AREA_{37-1MUX} - 11AREA_{16-1MUX} - AREA_{MAP} \\ AREA_{RF32} &= AREA_{SEL11} + 32AREA_{6-1MUX} + 32AREA_{gr} \\ &\quad + 11AREA_{37-1MUX} - 11AREA_{16-1MUX} - AREA_{MAP} \end{aligned}$$

The difference of the circuit area between RF32 and SEL11 depends on $AREA_{MAP}$.

It can easily be observed that

$$AREA_{RF11} > AREA_{SEL11}.$$

Despite of the high-density interconnection and a long delay in the RF32 model, it may have a smaller area if a necessary MAP in RF11 and SEL11 models has a dominant area. The cost in media registers can be similarly calculated as the general RF. A detailed area comparison between RF32, RF11+MAP and SEL11+MAP will be provided in Sect.4.

When there is no data dependency between loop iterations, LAPP can execute all instructions in a loop structure overlapping the succeeding iterations. This means the peak IPC of LAPP is equal to the number of instructions in a loop. Meanwhile, to cope with the self update instructions required for the maintenance of loop counters, additional bypass networks should be introduced. The self-updating loop counters and an incremental/decremental LD/ST address can be properly updated by forwarding its output value to its input to assure current execution. We use *self-forwarding* in later parts to denote this data forwarding. Note that this self-forwarding will be enabled from the second iteration in corresponding ALUs. The ALUs that corresponds to the self-forwarding get the initial source values from the register file for the first iteration, and get succeeding values from themselves for succeeding iterations.

3.2 Propagation Mechanism and Mapping Algorithm

The instruction mapper processes VLIW instructions in order and maps the i -th VLIW instruction onto stage i . Under our FU array pipeline structure, the architectural register value of a consumer instruction should be first forwarded from its latest producer instruction in the previous FU array pipeline stage. If no producer instruction of this architectural register is present in the previous stage, data propagation from further previous stages is required to provide correct inputs for the consumer.

A *propagation skip* table (denoted as *prop_skp* in later parts) is designed to indicate whether the propagation path from a previous stage is necessary or not. In the SEL model, each stage contains a *prop_skp* table, which is indexed by the architectural register number. All elements of this table are initialized to zero. From the viewpoint of the producer instruction in stage i , supposing that its destination is an architectural register p (p : register number), all *prop_skp*[p] in the stage from 0 to $(i+1)$ should be set, because the propagation of the value of p from stage 0 to i can be 'skipped' even if the succeeding instructions may refer to p . In this case, the succeeding instructions can get the correct value of p from stage i . As described, the *prop_skp* plays an important role in reducing the number of register propagation.

The instruction mapping algorithm with the *prop_skp* is shown in Fig. 5.

- (1) The mapping is started for VLIW instructions after a prefetch instruction.
- (2) i -th VLIW instruction configures every stage from stage 0 to stage $(i-1)$.
- (2-1) For all of the source registers in the VLIW instruction, if *prop_skp*[s] (s : source register number) is not one, one propagation register is allocated and the register s is renamed and assigned to the propagation register by *assign_prop_reg()* via *alloc_prop_reg()*.
- (2-2) For all of the destination registers in the VLIW instruction, register d (d : destination register number) does not have to be propagated to next stage. Thus, *prop_skp*[d] is set to one by *update_prop_skp()*.
- (2-3) The processing of self-updating is slightly complicated. At stage i , when we identify a self-updating instruction like *reg++* by detecting the same source and destination registers, all instructions that have the same source register from stage 0 to stage $(i-1)$ will be regarded as self-updating ones by *conf_self_fwd1()*. As an example, we have $i1$ as $R5+4 \rightarrow R6$ and lately have $i2$ as $R5++$. From the second loop iteration, $R6$ will be self updated as $R6++$.
- (3) i -th VLIW instruction configures stage i .
- (3-1) Instructions in the VLIW instruction are mapped onto execution units by *map_VLIW_inst()*.
- (3-2) Same step as (2-1) is applied by *alloc_prop_reg()* and *assign_prop_reg()*.
- (3-3) Same step as (2-2) is applied by *update_prop_skp()*.

Fig. 9 Step by step instruction mapping for (a) VLIW 0 and (b) VLIW 1.

Fig. 10 Instruction mapping steps for (a) VLIW 2, (b) VLIW 3 and (c) VLIW 4.

ecution units is completed. Note that Array-Setup is interrupted when the number of VLIW instructions in the loop kernel is larger than that of the stages. And Normal-Execution is restarted.

3.3 Implementation and Example

We describe the details of the interconnections in an LAPP and use sample code to illustrate the instruction mapping procedure on an LAPP. Figure 6 shows the modules around FUs in LAPP. VLIW indicates VLIW instructions, which are supplied from an instruction decoder in stage 0. The s1, s2 and s3 indicate EXEC input registers of each execution unit. The prop_skp denotes a propagation skip table. The wb is an EXEC output register of each execution unit. Figure 7 shows the interconnections of ALUs. EXEC input registers (s1 and s2) in stage i can get values from 11 EXEC input registers in stage $(i-1)$ and 5 EXEC output registers in stage $(i-2)$. Therefore, SEL in Fig. 2 contains 11 16-input 1-output multiplexers (16-1 MUX) and 11 EXEC input registers, which can be candidates for propagation registers. An execution unit in stage i can select its input data from its own input register, the 5 data forwarding paths from stage $(i-1)$

or its own EXEC output register, which corresponds to a self-update instruction.

Figure 8 (a), (b) and (c) show the sample program code, an assemble code generated from this program code and dataflow graph respectively. The program allocates array A, B and C, among which element A and B are input, element C is output. In VLIW 0 and VLIW 1, 2 values are loaded from A and B, as indicated by gr1 and gr4. When the condition flag z is set by a sub in VLIW 0, a bz (branch on zero) operation in VLIW 1 is executed and the loop kernel is completed. In VLIW 2 and VLIW 3, an intermediate value is calculated from gr2 and gr5 by a 16-bit sll (shift left logical) and an or operation. In VLIW 4, the result value is stored to C indicated by gr6. The loop kernel is repeated by a bra (branch always) operation in VLIW 4.

Figure 9 and Fig. 10 demonstrate an instruction mapping flow of the program as shown in Fig. 8. Among them, Fig. 9 (a) and (b) illustrate the mapping for VLIW 0 and VLIW 1 step by step, while the succeeding mapping results are shown in Fig. 10 (a), (b) and (c).

In Fig. 9 (a), VLIW 0 is mapped onto execution units in stage 0 at step 0. Since prop_skp[gr1] and prop_skp[gr3] are not one as shown in (3-2), gr1 and gr3 can be

mapped onto propagation registers. Following the rule in (3–3), $\text{prop_skp}[gr1]$, $\text{prop_skp}[gr2]$, $\text{prop_skp}[gr3]$ and $\text{prop_skp}[z]$ in stage 0 will be set to one. Using (3–4), $gr1+0$ will be self-updated as $gr1+=4$. Applying to (3–5), an execution unit for the **add** is configured with a self-forwarding. According to (4–1), since results of **add** and **sub** can be supplied to execution units in stage 1 by using a forwarding path from EXEC output registers, $\text{prop_skp}[gr1]$, $\text{prop_skp}[gr3]$ and $\text{prop_skp}[z]$ in stage 1 are set to one at step 1. According to (5–1), as a load data of **ld** can be supplied to execution units in stage 2, $\text{prop_skp}[gr2]$ in stage 1 and 2 is set to one at step 1 and step 2. Since there is no **bra** in VLIW 0 as shown in (6), and then VLIW 1 will be scheduled.

In the same way, in Fig. 9(b), since $gr4$ needs to be propagated as shown in (2–1) and (2–2), $gr4$ is mapped onto an EXEC input register of an empty execution unit in stage 0 at step 3. It is also mapped onto mapped execution unit in stage 1 at step 4. Furthermore, execution units for **add** and **ld** in VLIW1 are configured with their self-forwarding. At step 5 and step 6, $\text{prop_skp}[gr4]$ and $\text{prop_skp}[gr5]$ are set to one to forward results of **add** and **ld** to stage 2 and stage 3.

In Fig. 10(a), since $\text{prop_skp}[gr2]$ is one, $gr2$ does not need to be propagated at step 7 and step 8. It is forwarded from **ld** in stage 1 directly. VLIW 2, which is mapped onto stage 2 at step 9, uses a result of **ld** in stage 0. At step 10, $\text{prop_skp}[gr2]$ is set to one to forward a result of **sll** to execution units in stage 3.

In Fig. 10(b), since $\text{prop_skp}[gr2]$ and $\text{prop_skp}[gr5]$ are one, $gr2$ and $gr5$ do not need to be propagated at step 11 to step 13. They are forwarded from EXEC output registers in stage 2. VLIW 3, which is mapped onto stage 3 at step 14, uses the results of **sll** and **ld** in stage 2 and stage 0. At step 15, $\text{prop_skp}[gr5]$ is set to one to forward a result of **or** to execution units in stage 3.

In Fig. 10(c), $gr6$ needs to be propagated and is mapped onto EXEC input registers in stage 0 to stage 3 at step 16 to step 19. According to (2–3), execution units for **add** and **st** in stage 3 are configured with their self-forwarding at step 20. In addition, since **bra** is found in VLIW 4 as shown in (6), the instruction mapping of this loop kernel is completed.

The instruction mapping can be implemented by using either software compilers or hardware circuits. For software implementation, a compiler can generate configuration information. The information can be stored in configuration memory and read in Array-Setup. Alternatively, the configuration information can be generated in each stage step by step for VLIW instructions with pipelined instruction mappers. A mapped VLIW instruction can be stored inside a configuration FF and the other VLIW instructions that are propagated to the next stage are stored in a propagation FF. Each step in Fig. 9 and Fig. 10 can be done in the following execution flow.

CC 0: step 0 (VLIW0 is mapped)

CC 1: step 1, step 3

CC 2: step 2, step 4, step 7 (VLIW1 is mapped)

CC 3: step 5, step 8, step 11

CC 4: step 6, step 9, step 12, step 16 (VLIW2 is mapped)

CC 5: step 10, step 13, step 17

CC 6: step 14, step 18 (VLIW3 is mapped)

CC 7: step 15, step 19

CC 8: step 20 (VLIW4 is mapped)

The above flow indicates that several steps are executed at a clock cycle simultaneously. VLIW i is mapped onto stage i at clock cycle (CC) $2 \times i$. Therefore, the number of cycles for the configuration costs $2 \times N$ (N : the number of VLIW instructions) by using pipelined mapping.

4. Circuit Area and Delay Time

In this section, we quantitatively evaluate the circuit area and the delay time of the instruction mapper. First, to present the position of the instruction mapper, we compare the instruction mapper with each module in an LAPP by designing them with HDL and synthesizing them with Synopsys Design Compiler using a 180 nm process rule. Second, to prove relationships as described in Sect. 3.1, we designed 3 hardware models, RF32, RF11 and SEL11, as shown in Fig. 4. Third, to demonstrate the hardware cost for the instruction mapping scheme, we evaluate the circuit area of RF32, (RF11+MAP) and (SEL11+MAP). Finally, to present the hardware cost for one array stage, we evaluate the circuit area of each model. Each stage has the same configurations as described in Sect. 3.

We obtained the area and delay values by synthesizing under a relatively tight clock constraint—2 ns (500 MHz)—to find the best possible maximum operating speed of each component. Figure 11 and Fig. 12 show the delay time and the circuit area of each module under the tight constraint respectively. In Fig. 12, IF and LD/ST do not include the area of I1\$, L0\$, and L1\$. ALU, BRC, EAG and MEDIA indicate the total circuit area of each module. ALU and MEDIA contain unoptimized multipliers when using the synthesis tool. They can be implemented with custom multipliers to improve their delay time.

In Fig. 11, the delay time of SEL11, RF11 and RF32 are 5.6 FO4, 6.0 FO4 and 7.0 FO4 respectively. Meanwhile, the delay time of MAP is 12.7 FO4, which is the third-longest in the modules of LAPP after ALU and MEDIA. Since the MAP only works during Array-Setup, the execution time of MAP does not affect Normal-Execution and Array-Execution. For example, if MAP has 2 cycles for mapping an instruction to shorten its delay time, the number of cycles of the configuration becomes $4 \times N$ (N : the number of VLIW instructions). The execution time of the instruction mapping can be hidden under data prefetching as described in Sect. 2.

Figure 12 shows that the circuit area of RF32 which includes many traditional registers and multiplexers is the largest in LAPP. Diverting EXEC input registers, Our proposed SEL11 has the smallest circuit area and the shortest delay time of the evaluation models by the proposed mapping scheme. From the delay time of MEDIA, a valid clock constraint can be set to 6 ns (166 MHz).

Fig. 11 Delay time of each module under a tight clock constraint.

Fig. 13 Delay time of each module under a validity clock constraint.

Fig. 12 Circuit area of each module under a tight clock constraint.

Fig. 14 Circuit area of each module under a validity clock constraint.

We also investigated the current design by synthesizing under this 6 ns constraint. Figure 13 and Fig. 14 show the delay time and the circuit area of each module under the validity constraint respectively.

Figure 13 shows that all modules can meet the timing constraint. There are many discussions about the optimal number of FO4 per pipeline stage. A research by Hrishikesh, et al.[11] indicated that an optimal delay for one stage is around 6 to 8 FO4. Viji Srinivasan, et al.[12] showed that an optimal design point based on a power-performance metric ((Billions of Instructions Per Second)³/Watt) is 18 FO4 per pipeline stage. It is our future work to find optimal depth of pipeline stages for LAPP.

In Fig. 14, the circuit areas of RF32, SEL11, RF11 and MAP are 87.8 K gates, 31.3 K gates, 41.0 K gates and 32.9 K gates respectively. The circuit area (MAP+SEL11) for the instruction mapping scheme is 84% of RF32. We also evaluated the hardware cost for one array stage in RF model and SEL model. The hardware cost of RF32 model and SEL11 model are 196 K gates and 172 K gates respectively. The total hardware cost of SEL11 model including both the instruction mapping and array stage is 88% of the RF32 model.

5. Related Work

CGRA is one major direction to obtain better power consumption performance by properly reconfiguring the interconnections between many function units (FUs). Many researches have been pursued by giving different CGRA implementations to exploit parallelism. The major differences reside in the unit interconnection, memory accessing and program inputs.

VLDP [13] and LSRDP [14] works on a dataflow graph by using an ALU-Net engine, which contains ALUs or a two dimensional processing element (PE) array respectively. Both of them require specific compiler supports to generate the configuration information. In addition, input data to the working units of these two accelerators is supplied from a limited number of LD/ST units. The shortage of LD/ST units will usually be the bottleneck for the whole system.

TRIPS [8] supports the Explicit Data Graph Execution (EDGE) [15] instruction set architecture (ISA), in which instructions are designed to explicitly indicate the operations and the connections between these operations. As demonstrated from their results, TRIPS highly depends on the efficiency of the compiler to generate perfect dataflow controls. Therefore, it lacks compatibility with conventional processors.

ADRES [3] is composed of a VLIW engine and an accelerator engine. The VLIW engine is used to support a VLIW-like programming model for legacy code. The accelerator engine accelerates loop kernel execution within a two dimensional array of coarse-grained PEs. Though ADRES shows better programmability than the above CGRA implementations, program recompiling by a specific compiler is still necessary to execute the program on the accelerator engine.

As an alternative to the above architectures, LAPP employs level 0 data caches to handle many memory access requests in a loop kernel simultaneously. To maintain consistency of level 0 data caches with small bandwidth, LAPP focuses on spatial data locality between array stages. Furthermore, to achieve backward compatibility with traditional processors, LAPP is designed to take existing programs that include prefetch instructions as input to a VLIW pipeline. After reconfiguring the interconnection, an FU array pipeline can exploit maximum parallelism between loop iterations.

PPA [6] is designed with flexibility to enable hardware to be dynamically customizable to the applications. PPA can map a dataflow graph onto the different PE arrays by using its virtualized modulo scheduling in a special compiler. However, as the PE array grows larger in size, the scheduling becomes more complex. Our proposed scheme provides high-scalability for array stages by linearly extending the backend pipeline of the VLIW pipeline.

The proposed instruction mapper is a key component to achieve this LAPP accelerator. It works on the general input of the VLIW pipeline and tries to generate a suitable mapping/interconnection to achieve array execution.

6. Conclusion

We proposed an instruction mapping scheme to execute traditional VLIW instructions on our low-power FU array processor named LAPP. The instruction mapper maps VLIW instructions onto the FU array and configures its interconnection. To reduce the circuit area and the delay time, the instruction mapper uses register renaming from architectural register to EXEC input register.

We evaluated the circuit area and the delay time of the instruction mapper by designing with HDL. Focusing on the same function, the circuit area for the proposed model is 84% of the baseline model. We also evaluated the hardware cost for one array stage in each model. The instruction mapping scheme achieves a similar functionality with 88% of the hardware cost of one array stage with the baseline model. We conclude that the instruction mapper provides appropriate circuit area for instruction mapping.

In future work, based on a detailed implementation of LAPP with the proposed mapper, we will verify the efficiency by comparing with other alternative architectures such as CGRAs and many-core architectures.

Acknowledgements

A part of this work is supported by Adaptable and Seamless Technology Transfer Program through Target-driven R&D, Japan Science and Technology Agency, and Grant-in-Aid for Young Scientists (B) 22700053. This work is also supported by VLSI Design and Education Center (VDEC), University of Tokyo in collaboration with Synopsys, Inc. and Cadence Design Systems, Inc.

References

- [1] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.C. Miao, J.F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol.27, no.5, pp.15–31, 2007.
- [2] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pp.1–15, ACM, New York, NY, USA, 2008.
- [3] B. De Sutter, P. Coene, T. Vander Aa, and B. Mei, "Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays," *LCTES '08: Proc. 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pp.151–160, ACM, New York, NY, USA, 2008.
- [4] T.G. Noll, T. von Sydow, B. Neumann, J.S. Schleifer, T. Coenen, and G. Kappen, *Reconfigurable Components for Application-Specific Processor Architectures*, pp.25–49, Springer Netherlands, 2010.
- [5] S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor, and R. Laufer, "PipeRench: A coprocessor for streaming multimedia acceleration," *International Symposium on Computer Architecture*, pp.28–39, 1999.
- [6] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications," *MICRO 42: Proc. 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp.370–380, ACM, New York, NY, USA, 2009.
- [7] H. Singh, G. Lu, E. Filho, R. Maestre, M.H. Lee, F. Kurdahi, and N. Bagherzadeh, "Morphosys: Case study of a reconfigurable computing system targeting multimedia applications," *DAC '00: Proc. 37th Annual Design Automation Conference*, pp.573–578, ACM, New York, NY, USA, 2000.
- [8] M. Gebhart, B.A. Maher, K.E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robotmili, A. Smith, J. Burrill, S.W. Keckler, D. Burger, and K.S. McKinley, "An evaluation of the trips computer system," *ASPLOS '09: Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.1–12, ACM, New York, NY, USA, 2009.
- [9] T. Nakada and Y. Nakashima, "Adaptability of a linear array vliw processor (in japanese)," *Hokkaido High Performance Computing and Architecture Evaluation Workshop (HOKKE2009) 2009-ARC-186*, no.10, pp.1–9, 2009.
- [10] T. Iwakami, M. Agari, K. Yoshimura, T. Nakada, and Y. Nakashima, "Area optimization of fu array in low-power accelerators," *IEEE Symposium on Low-Power and High-Speed Chips 2010 (poster)*, p.194, 2010.
- [11] M.S. Hrishikesh, D. Burger, N.P. Jouppi, S.W. Keckler, K.I. Farkas, and P. Shivakumar, "The optimal logic depth per pipeline stage is 6 to 8 fo4 inverter delays," *ISCA '02: Proc. 29th Annual International Symposium on Computer Architecture*, pp.14–24, IEEE Computer Society, Washington, DC, USA, 2002.
- [12] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P.N. Strenski, and P.G. Emma, "Optimizing pipelines for power and per-

formance,” MICRO 35: Proc. 35th Annual ACM/IEEE International Symposium on Microarchitecture, pp.333–344, IEEE Computer Society Press, Los Alamitos, CA, USA, 2002.

- [13] H. Irie, N. Hattori, K. Yamaguchi, Y. Tanaka, S. Sakai, and H. Tanaka, “Vldp3: Large scale architecture which enables high speed data-flow execution (in japanese),” Software and Hardware Integration for Innovative Next Generation Computing (SHINING2003) 2003-ARC-151, no.10, pp.49–54, 2003.
- [14] F. Mehdipour, H. Honda, H. Kataoka, K. Inoue, and K. Murakami, “An accelerator based on single-flex quantum circuits for a high-performance reconfigurable computer,” Workshop on Accelerators for High-Performance Architectures (WAHA09), 2009.
- [15] D. Burger, S.W. Keckler, K.S. McKinley, M. Dahlin, L.K. John, C. Lin, C.R. Moore, J. Burrill, R.G. McDonald, W. Yoder, and t.T. Team, “Scaling to the end of silicon with edge architectures,” Computer, vol.37, no.7, pp.44–55, 2004.

Kazuhiro Yoshimura received the B.E. degree from Ryukoku University, Japan in 2007. He received the M.E. degree from Nara Institute of Science and Technology, Japan in 2009. He is currently a doctoral student in Graduate School of Information Science, Nara Institute of Science and Technology. His graduate research includes an energy efficient SMT processor with heterogeneous instruction architectures and an instruction decomposition method for reconfigurable decoders. He is a student member

of IEEE and IPSJ.

Takuya Iwakami received the B.E. degree from Kyushu Institute of Technology, Japan in 2009. He is currently a master's student in Graduate School of Information Science, Nara Institute of Science and Technology. He is a student member of IPSJ.

Takashi Nakada received the B.E., M.E. and Ph.D. degrees from Toyohashi University of Technology, Japan in 2002, 2004 and 2007 respectively. Since 2007, he has been an assistant professor in the Graduate School of Information Science, Nara Institute of Science and Technology. His research interests include processor architecture and related simulation technologies. He is a member of ACM and IPSJ.

Jun Yao received the B.E., M.E. degrees from Tsinghua University in 2001 and 2004 respectively. He received the Ph.D. degree from Kyoto University in 2009. Since 2009, he has been an assistant professor in Graduate School of Information Science, Nara Institute of Science and Technology. His current researches are in computer architecture field. He is a member of IEEE and IPSJ.

Hajime Shimada received the B.E., M.E. and Ph.D. degrees from Nagoya University in 1998, 2000 and 2004 respectively. From 2004 to 2008, he was an assistant professor in the Graduate School of Informatics, Kyoto University. Since 2009, he has been associate professor in the Graduate School of Information Science, Nara Institute of Science and Technology. His research interests are processor architecture. He is a member of IPSJ.

Yasuhiko Nakashima received the B.E., M.E. and Ph.D. degrees in Computer Engineering from Kyoto University, Japan in 1986, 1988 and 1998 respectively. He was a computer architect in the Computer and System Architecture Department, FUJITSU Limited in 1988–1999. From 1999 to 2005, he was an associate professor in the Graduate School of Economics, Kyoto University. Since 2006, he has been a professor in the Graduate School of Information Science, Nara Institute of Science and Technology. His

research interests include processor architecture, emulation, CMOS circuit design, and evolutionary computation. He is a member of IEEE CS, ACM and IPSJ.